International Journal of Engineering Research and Technology. ISSN 0974-3154, Volume 12, Number 8 (2019), pp. 1239-1246
© International Research Publication House. http://www.irphouse.com

Visualization and Synchronization of Object-oriented Programs using Re-engineering Approach

Ahid Yaseen
Department of software engineering
Zarqa University, Zarqa, Jordan.

Hamed Fawareh
Department of software engineering
Zarqa University, Zarqa, Jordan.

Abstract

Software visualization considered as an essential approach in software maintenance. This visualization approach used to increase system understandability. In this paper, we proposed a Maintenance Visualization and Synchronization Tool (MVST). The tool developed to visualize and synchronize the object-oriented program during the maintenance process. MVST combine several tools to achieve their objectives. The first one, java CC &Coco/R to transfer the code written by Java and C++ to Abstract syntax tree and textual input. Second tool CSSFormatter to produce the scalable Vector Graphics (SVG) form the Textual Input. MVST extract the information from SVG and send the extracted data to the third tool PlantUML based on setting rules to produce a MetaModel. At this stage, we allow maintainer to modify on the UML diagram. The fourth tool JAD to decompile the modified code. MVST constructs diagrams from existing Object-Oriented code in both static and dynamic forms. Maintainer changes the static and dynamic diagrams to reflect directly to the source code.

Keywords: software visualization, software reengineering, software maintenance

1.0 INTRODUCTION

Large applications build thousands Line Of Code (LOC), which have a huge number of classes and complex components. That makes maintaining software more hard and complicated, precisely without written documentation, and lack standardization of code. Also, the code understanding will be a hard and consuming effort.

Object-Oriented Programs contains huge dependencies and various relations between modules. Nevertheless, the relationships have multi types of associations between classes and objects. These associations mentioned as inheritance, realization, composition, and aggregation. Each one has a multiplicity of direct association or reflexive one [1].

Maintained application has enormous numbers of relations. That is undoubted, make relationships discovery a bit hard on maintainers and with difficulty, especially when do it manually. Thus, an auto visualization tool is a better choice in regards to saving times and improved coverage.

Most of the researchers in our literature reviews concern on developing a tool to help maintainer on the comprehensive and understanding source code. However, build visualization, and synchronization tool is challenge research for all UML diagram, in light of unstructured programming. Furthermore, only 7% of visualization researches used efficient evaluation method [2]. Moreover, Raibulet et al. [3] mentioned only 20% of reverse engineering researches concern on UML Models visualization as a maintenance process.

Integrated the static and dynamic visualizations during maintenance OOP environment is a motivation in this research; thus, we proposed a Maintenance Visualization and Synchronization Tool (MVST), which supports Reengineering concepts with high comprehension with both static and dynamic dependencies.

This paper focuses on generating a visualization and synchronization maintenance tool. The approach of this paper will help to increase the rate of comprehension software and better program understanding. The proposed tool aims to reduce maintenance effort and cost. Moreover, make software enhancement more efficiently to be suitable with new requirement during the maintenance phase. The developed tool in this paper will achieve: reduce efforts and costs consumes in the maintenance phase, enhance the perfective modification process by generating appropriate synchronized tool and make automated re-engineering process more efficient.

2.0 RELATED WORKS

In recent years, there has been an increasing amount of literature on reengineering and the effect on maintenance. Various approaches to reverse engineering and software visualization, concern on static and dynamic transformation.

Martinez et al., [5] used RE to visualize source code to sequence diagrams. Based on source code visualization, the researchers created a class model as an initial phase and then converted it to sequence diagram via Object Constraint Language (OCL). These models help describe the dynamic structures only. The researchers did not employ a full reengineering concept but only created a visualization source code without synchronizing with a sequence diagram.

Améndola [6] and Bruneliere et al.[7] proposed model-driven reverse engineering for representing the legacy systems called Model Discovery (MoDisco). This model based on two steps start with detecting and finding primary artefacts in legacy systems, then understanding and creating models. MoDisco applied into java, JEE, and XML platform. The proposed idea centres on just visualization the source code. Hence, it has a shortage of alteration and refinement phases on the re-engineering of the source code.

Garzón et al., [8] visualized the Object Oriented (OO) into UML called using a tool called Umplification. Umple is an open source code support, C++, PHP, Java and Ruby languages. The tool concern with refactoring system, by an incremental stepwise transformation of source code. The Umple approach is an increment approach does not deal with a large system. Umple visualizes source code to state machine only does not involve Static and dynamic structures.

Fawareh [9] developed an approach to solving the same problem, but without completing the reengineering concepts. The approach taken divides reverse engineering into three significant levels. First one uses class level: extract the classes and relationships — second level concerned with method level: extract method association with themselves and with variables. The last turn focuses on the lower level, statement: explain association types. This approach did not provide synchronized visualization.

Synchronized UML diagrams approach in many programming; web-based and in OO programming. Several tools used to support static and dynamic forms. Then merge dynamic and static. Subsequently, transfers source code to both views using suggested tools called PlantUML, Java visualizer, JaguarCode and Javelina. The differences between these tools in programming language and environment [10].

Merging both forms static and dynamic into one tool is a good idea. However, this research focused on representing class and sequence diagrams for educational purposes. Also, that simple tool could not detect any errors in the maintenance phase.

3.0 MAINTENANCE VISUALIZATION AND SYNCHRONIZATION TOOL (MVST)

This research proposes Maintenance Visualization and Synchronization Tool (MVST). Essentially, MVST visualizes the Object Oriented Programming (OOP) source code to both Unified Modeling Language (UML) structure views: static and dynamic. Furthermore, MVST allows the maintainer to enhance the program directly to static UML Diagrams. Subsequently, any modification for enhancement will concurrently reflect directly to source code and vice versa, this means to generate a new system as a Forward Engineering phase.

As the Reengineering concept, MVST has three main activities: Reverse engineering, Alteration and Forward Engineering. Figure.1 shows the research methodology.

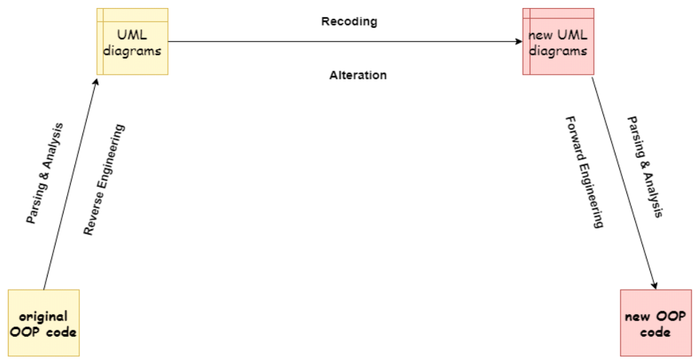


Figure 1: MVST methodology

Current researches concerning on visualizing single view of UML and does not involve synchronization process. Moreover, all of them visualize the structured code only. Thus, the MVST proposes to achieve the following goals:

- The various format of visualization of the standard and unstandard system. To help the maintainer to understand source code, and increase programming comprehension by supporting the two structures of UML view: static and dynamic. Starting from each line of code that executes, to the whole classes and methods.
- To create a chance to keep the maintenance phase and program enhancement rapidly and efficiently.
- To re-engineer the modification in multiprogramming language considering OOP.

3.1 MVST Approach

MVST applies the reengineering method on one part of the system only, and then it integrates the non-engineered part of the system. Thus, MVST follows partial reengineering approach.

The partial approach is divided into three main steps as follow:

- Firstly, the maintainer has to split the current system into many parts: Only one part has to re-engineer, and others have to re-engineer lately.
- Secondly, MVST performs the reengineering work using the Big Bang .
- Thirdly, applies the integration between the components, to produce the new system.

- **External Tools**

MVST utilized external tools with some modification in their source code to deal with MVST contribution, which has a significant role in supporting the research and facilitate the implementation process. Similarly, it is most flexible and has a shorter development time. This section explains all the tools and describes its importance.

- **Coco/R**

Coco/R is a parser and compiler generator considerable with several languages including Java, C# and C++. Furthermore, Coco/R converts the code to text, to be readable via JavaCC parser. This text translates from several OO programming language [11].

MVST hire Coco/R at the first phase based on its job as virtual machine parser. That deals with JavaCC parser to support various languages in the visualization process.

- **JavaCC Parser**

JavaCC is "an open-source analyzer and parser generator and lexical analyzer generator written in the Java programming language" [12].

Besides, JavaCC splits the code into tokens based on some formal grammars. JavaCC also generates lexical analyzers. Moreover, JJTree includes in JavaCC that is tree builder.

MVST using JavaCC for parsing and analyzing OO source code to textual input, by removing comments and other minor parts to be able to convert it to Scalable Vector Graphics (SVG).

- **PlantUML**

PlantUML is a web-based open-source visualization tool that supports Reverse Engineering to create UML diagrams from a plain textual description in simplified Scalable vector graphics (SVG) form [13].

PlantUML can generate several UML diagrams from structure code only: Sequence and State machine diagrams as a dynamic structure. Also, Class, Use-Case and Object diagram as a static structure.

PlantUML defines some rules and keywords to generate class diagram and realize classes and relations from structured OOP only. For instance, it searches about "interface" keyword for defining Interface class. As well, "abstract, static" keyword indicates the type of class and method. Nevertheless, for OOP concepts like inheritance PlantUML searches about "extend" keyword, etc.

- **Customized PlantUML**

Customized PlantUML: For the primary purpose, we made some changes to be able to visualize unstandard code too. To achieve this point we setting out our own new rules, such as adding, inserting new colourful diagrams options and defined the candidate dependencies. These rules and theory explained in MVST model.

- **NBJAD**

NBJAD (NetBeans Java Decompile) is plug-in source codes integrate JAD DEcompilation with NetBeans for java code. De-compiler means of "taking an executable file as input, and attempts to create a high-level source file that can be recompiled successfully" [14].

MVST needs de-compiler at refinement phase for decompiling the executed code to the source code before de-compilation directly.

3.2 MVST Model

This section explains the proposed MVST model. MVST is partially automated tool concern on perfective maintenance and programming comprehensive. Hence, the maintainer has to involve the modification process. Figure 2 shows the MVST process.

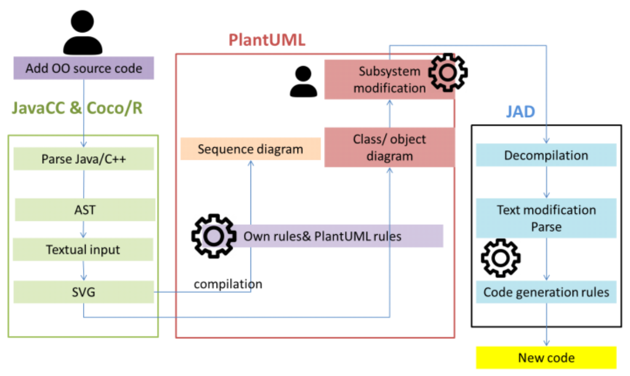


Figure 2: MVST process

MVST divides the task into specified steps, shown in details as follows:

Firstly, Coco/R parses the source code written in Java/C++ or ruby language syntax to the Virtual Machine (VM). The Coco/R parser is working on the source code by transforming it to Abstract Syntax Tree (AST). Intending to access several programming languages and including a large number of applications.

Secondly, JavaCC analyzes the AST, to view the important information and exclude all unimportant semantics, for instance, comments. This output is called Textual Input.

Thirdly, the CSSFormatter tool produces the Textual Input to Scalable Vector Graphics (SVG). That is needed to utilize an external tool.

Fourthly, MVST sends SVG to External tool called PlantUML that receives the SVG and extract the classes, objects and elements based on some specific rules integrated by this research rules and PlantUML MetaModel.

MVST Not only draw and view the static diagrams but also, will represent Dynamic views when SVG compiled.

The visualized views synchronized with the original source code. Thus, any modification on static views has to refine again with rules. The main challenges face MVST is to modify compiled code, so MVST using JAD to decompile the codes.

- The first step on refinement is to parse the output. The NBJAD will analyze and transform models into syntax forms.
- Eventually, MVST will generate source code again

Class Diagram Visualization: class diagram consists of necessary elements to display all relations and dependencies. This part demonstrates how to convert SVG to the class diagram in particular.

In principle, MVST needs to define specific rules based on definitions, our experience in coding and the literature review for extracting and producing a class diagram, the rules set as follows:

Classes Relations: The class is a set of objects created as a group. Each class connects with others through links that have different types of relations. This rule of this relation is describe as follow:

Case1:

$$(\exists C1.v\ name == C2\ name \vee \exists C2.v\ name == C1\ name) \rightarrow C1, C2\ has\ association$$

Case2:

$$\exists\ C1\ obj1 = new\ C1(..) \wedge C2\ obj2 = new\ C2(\dots) \rightarrow C1, C2\ has\ an\ association$$

Case3:

$$\forall\ interface\ C1\ \exists\ implements\ C2 \quad \therefore C1, C2\ has\ association$$

Case4:

$$\exists\ interface\ C\{..Class.m1()\}$$

$$C1.obj.m1(C2.obj)$$

$$\therefore C1, C2\ has\ an\ association$$

- Dependency (Cause-Effect Association): indicates that changes to one class can cause changes in another class.
  $$\exists\ C2\ obj2 = new\ C2\ \&\ C1.M1(obj2) \rightarrow C1\ depend\ on\ C2$$
- Aggregation: describe a class as a part of, or as underlying to, another class. C1 is part of C2. However, if C2 deleted that is not necessary to delete C1.
  $$\exists\ C1\{\ private\ C2[]\ Array\ =\ new\ C2[10]\ \dots.\}$$
- Composition: identify the lifetime of the part classifier is contingent on the lifetime of the whole class.

$$\exists\ C1\{\ constructor()\{final\ C2\ obj2{=}new\ C2()\}\}$$

∴ C1 owns the responsibility of creating the C2.

- Generalization: that a specialized (child) class based on a general (parent) class.
  $$\exists C2\ (\forall\ C1.m\ \&\ C1.v)$$
  $$\therefore C2\ \subseteq C1$$

Explaining the prototype in more details and mentions the reasons for limitation.

Static Diagram Modification

After visualization, the modification process starts. we describe the modification types, which the maintainer could do on the class diagram, as an Alteration activity. Figure 3 shows the operation on a class diagram.

Adding Process

Virtually, adding new features is too essential in perfective maintenance, where the maintainer adds new variables or methods. MVST provides this addition at the class diagram level to facilitate the maintenance phase. That is to say, the class diagram in MVST implement as a Stack Data Structure,

Deleting Process

In practice, maintenance maintainer rarely needs to eliminate some features. MVST provides trait of deleting some features, which depends on maintainer experience. This modification includes variables, methods or the whole class. After that, MVST pops the determined item out of the stack. Virtually, the deleted object not really deletes, but only masks the relations temporary.

- Determine the method to delete
- Tracing all related parts of coding (show them at real time)
- Display a Pop-up message to ensure the maintainer intended the action
- Mask the methods as comments
- After a while, if the maintainer does not need a method and all related parts will delete. This feature used traceable algorithms to achieve this goal.

Changing Process

There are many simple changes needs in maintenance. Markedly, changes include modifying types or multiplicity. Thus, MVST firstly searches about the variable that required changing type, secondly viewing any confused types or data, afterwards, MVST modifies the selected type, and surely, a modification will be fulfilled after it is agreed upon asking the maintainer.

Renaming Process

Rename classes and variables more efficient when many maintainers named classes with a different name. MVST provides rename feature to simplify the correction process.

Dynamic Structure Visualization

Dynamic structures include several diagrams. MVST decided to visualize sequence diagram because of its importance during the analysis and design of OO systems.

Sequence Diagram Visualization

The sequence diagram is one of the dynamic structures, which means to visualize after compiling the SVG of source code. Initially, a sequence diagram concerns on an interaction between object and system involving the time.

MVST read internal code specifically the operation's body so that MVST could recognize the method invocations. Then drawing sequence diagram depends on the invocations result at runtime.

For more details, sequence diagram represents one method, and his calls act as messages. Moreover, the dependencies between classes represent in sequence diagram too.

Object's Relations:

$(if\ C2\ implements\ interface\ C1) \wedge (\exists\ \{C1\ obj{=}new\ C2();\ C1.m2(message)\}\ ) \leftrightarrow obj1\ interact\ obj2$

Figure 4.7 below illustrates a sequence diagram and shows the Activation bar. Activation bar brings to light the life of object which still active during messaging.

Message name: $\forall\ \boldsymbol{Att{=}message\ name}$

Synchronous message: occurs if the sender sent message to receiver one and block or waits to process. $\exists(if\ C2\ implements\ interface\ Clis) \wedge \exists\ C1\ obj = new\ C1();\ Clis\ obj = new\ C2();\ obj.Clis\ m\ (obj);\ obj.C1\ m\ ();\ \therefore \boldsymbol{message\ is\ Synchronouns}$

Asynchronous message: the calling process does not block the program from the code execution. $\exists\ new\ Thread(new\ Runnable() \rightarrow message\ is\ Asynchronouns$

Reflexive message: object send message to itself. $\exists\ object.m(this) \rightarrow obj\ send\ to\ itself$

Object Diagram Visualization

Object diagram "An object diagram is a graph of instances, including objects and data values. A static object diagram is an instance of a class diagram; it shows a snapshot of the detailed state of a system at a point in time. The use of object diagrams is fairly limited, namely to show examples of data structure" [15].

Object diagrams will visualize after the compilation process. The object diagram is related to the class diagram, so this needs to create classes first then create an object diagram.

Object Names: $\forall\ C1 \rightarrow \exists\ C1\ obj{=}new\ C1(); \leftrightarrow obj{=}{=}obj\ name$

Or, $\exists\ Arr[]\ obj{=}new\ Arr[] \rightarrow obj{=}obj\ name$

Object Attributes: $\forall\ C.v = obj\ Att \sim C.static\ v \notin obj\ Att$

At programming runtime, every value of an object sign to obj. Att is set as the value. Also, all dependencies between objects are related to class diagram dependencies that are already mentioned before.

4.0 MVST Application

MVST aims to increase the quality and efficiency of the maintenance process, which provides high comprehension of source code by visualizing the program, to facilitate program understanding. Therefore, we need to develop MVST to prove the research theory and observe any effects of the proposed tool.

Significantly, developing a full version of MVST will take a time. Nevertheless, it will be costly as it needs huge effort and professional team. Thus, the researcher decided to limit the implementation of building a prototype focusing on one part only of the whole model.

Furthermore, in light of choosing the best-selected part, the researcher looked for the most essential and significant diagram that directly affects the program understanding and views software structure. This research found that the best choice to consider in this research was to manipulate the class diagram because of its positive impact on maintenance. Thus, to employ a case study approach the researcher chose to visualize and reengineer the class and sequence diagram.

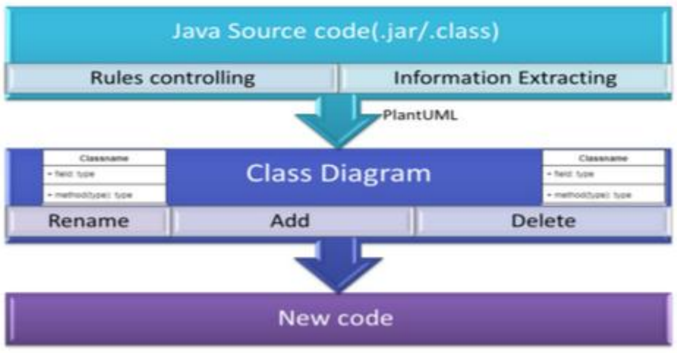


Figure 3: MVST system overview

MVST Platform: MVST is being developed on NetBeans Integrated Development Environment (IDE), which is written in Java programming language. Java is selected based on the platform independently. In another word, the MVST java source code can run on all operating systems. The target users of MVST are object-oriented maintainers, for valuable understanding. Also, maintainers will use it for modification purpose. Figure 4 shows flowchart for Visualization phase includes Classes' Relation determining.

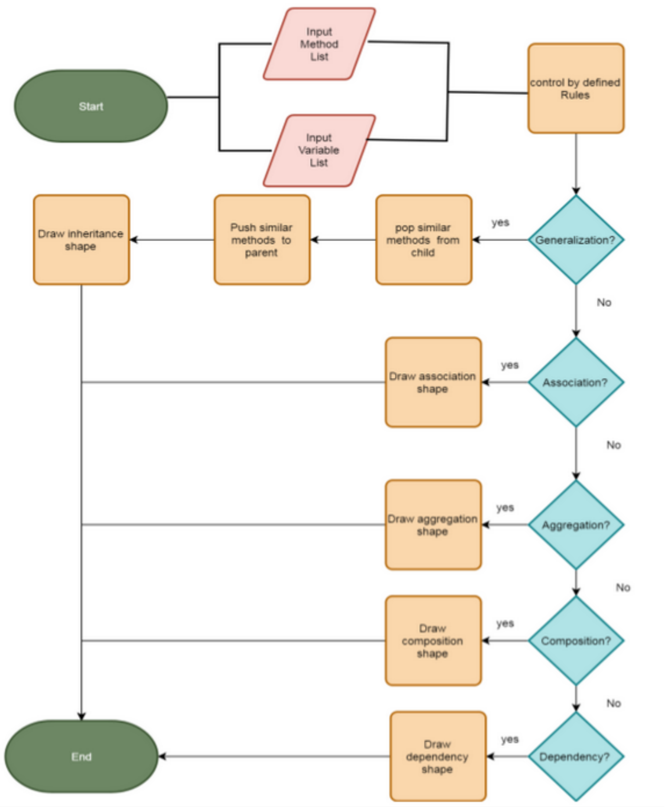
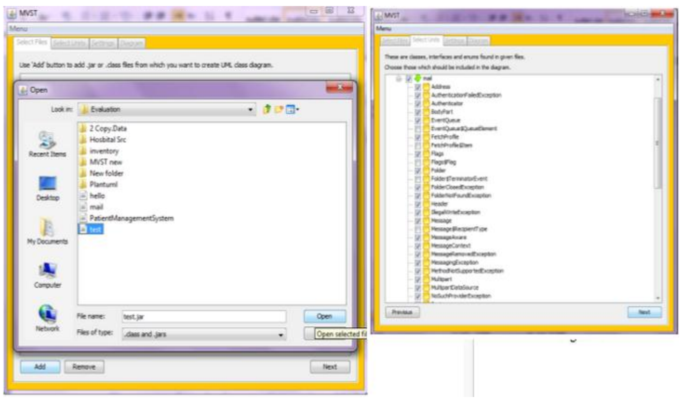


Figure 4: Relations Flowchart


Use 'Add' button to add .jar or .class files from which you want to create UML class diagram.

These are classes, interfaces and enums found in given files. Choose those which should be included in the diagram.

Case Study

An inventory management system is a system that consists of both software and hardware. This system track items and goods, including each process from production to warehousing to shipping.

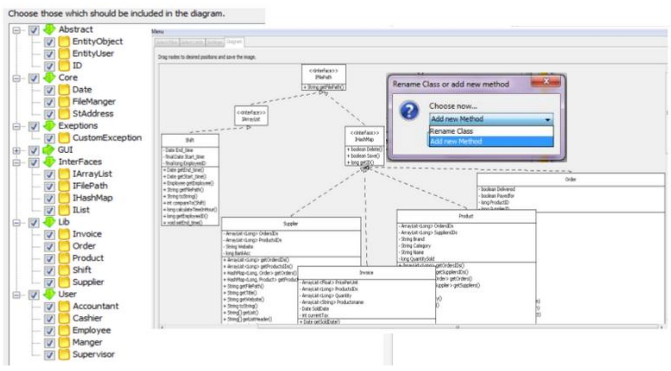
Inventory management system plays a significant role in business, which analyzes the entered data, predicts the future demands, pricing and creates frequently reports.

Furthermore, there are many advantages of using inventory management system: start with providing the best organization of costs and cash flow. Nevertheless, it is an excellent way to increase work efficiency and expand up to date data.

On the other hand, an inventory system is very complex structures and expensive systems.

To resolve the comprehension issue, the MVST has chosen to visualize the system to a class diagram for the whole system and provide a chance for maintainers to select a short method of the class diagram. Currently choosing the quick process of what needs maintenance allows the maintainers to understand it quickly.

Then, the process addition methods of the MVST tool as well as the class renaming done by the maintainer. Figures 6.1-6.5 below show classes diagrams of multi methods of inventory system.


Choose those which should be included in the diagram.

Drag nodes to desired positions and save the image.

MVST Evaluation

To evaluate MVST, the researchers presented the tool and two selected case studies and the points of views of software maintenance experts. There are many characteristics of the chosen experts to start with their background in their software engineering, their experience in software maintenance and java coding. In this research, the researcher followed Nielsen heuristic evaluation (Nielsen, 1994)

Therefore, the choice of an expert's set is divided into three Groups:

"Group 1" includes maintainers of selected case studies. This group had to use MVST in the comprehension compared to do it manually. Moreover, they evaluate MVST features and assess the MVST relation's suggestion. Table 1 below expresses Group 1.

Table 1: Group 1

Group 1	Gender	Programming	Development Years	Maintenance background	Education	Degree
Participant 1	F	Java	5	intermediate	C.S	Master
Participant 2	M	Java	6	high	S.E	BA
Participant 3	M	Java	5	Low	C.S	BA
Participant 4	M	Java	8	intermediate	S.E	BA

*C.S: Computer Science, S.E: Software Engineering, BA: Bachelor.

"Group 2" concerns maintainers who have a master degree in software engineering and their knowledge on development for more than one year. They had to evaluate MVST depending on maintenance principles and its effectiveness. Table 2 indicates Group2 information

Table 2: Group2

Group 2	Gender	Programming	Development Years	Maintenance background	Education	Degree
Participant 1	M	C++	2	intermediate	S.E	Master
Participant 2	M	Java	4	high	S.E	Master
Participant 3	F	Java	2	low	S.E	Master

"Group 3" interests on java maintainers that have a good background in maintenance and more than two years of programming experience. This group had to assess MVST Design as an End-user from the usability side. See Table 3.

Table 3: Group3

Group 1	Gender	Programming	Development Years	Maintenance background	Education	Degree
Participant 1	F	Java	2	Low	S.E	M
Participant 2	M	Java	2	Low	S.E	BA
Participant 3	M	C++	2	Low	C.S	BA
Participant 4	M	Java	3	Low	S.E	BA
Participant 5	F	Java	4	Low	S.E	M

The total number of participants is twelve persons that are good depends on Nielsen evaluators numbering recommendations.

At the evaluation training process, the participants give tasks based on what they had learnt. Furthermore, there are two categories of systems used during tasks; small systems have 2-4 KLOC, and medium systems have 10-14 KLOC. Table 4 shows the tasks given to participants:

Table 4: Participants' Tasks

Task No.	Manually	Using MVST
1	Describes their understanding of the short method	Describes their understanding of the short method
2	Describes classes relation of short method	Describes classes relation of short method
3	Mentions all child of parents	Mentions all child of parents
4	Renames class, variable, method	Renames class, variable, method
5	Adds class, variable, method	Adds class, variable, method

There is a positive relationship between a maintenance experience and a cognitive of reengineering importance. In this research, the researchers asked participants to assess MVST usefulness. The high experience in maintenance makes the assessment more transparent.

Figure 4 displays the relationship between the maintenance experience and the MVST evaluation. Where number 5 is the highest value of assessment and number 0 is the lowest value. Y-axis cross maintenance years, while X-axis cross participant number.

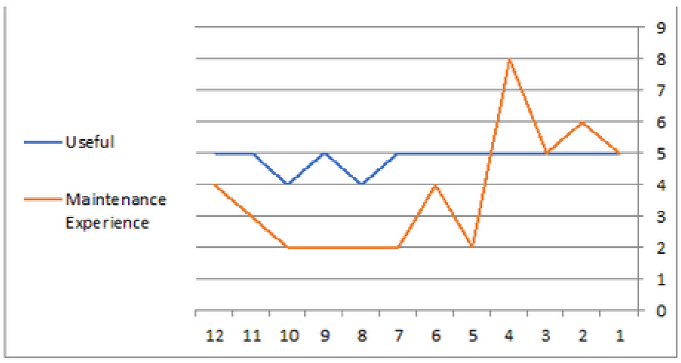


Figure 4: MVST Usefulness and maintenance experience

The chart expresses that MVST is very useful and has a significant role in maintenance phases. Nevertheless, most participants who have high expertise believe that visualizing the code into the class diagram and reengineer have a significant influence in software maintenance.

MVST Usability

Usability of the MVST focuses on four types: easy to use as a first time, easy to learn after training on the tool, number of steps acceptance needs to visualize and reengineer code and number of helps acceptance.

The participant's assessment shows the easy to learn of MVST. 91% of participants evaluate MVST as easy to determine where value 5 is the highest value of outstanding, 9% assess MVST as value 4, and nobody assesses 0/1/2/3 benefits. This data indicates a high level of easy to learn.

While MVST easy to use evaluation presents as follows: value 5 occupies 83.3% of the total assessment, while 16.7% of participants assess value 4 for easy to use, and 0% of participants choose another value.

The number of steps acceptance has a significant impact on the End-user impression. The result indicates the high satisfaction of participants about the number of steps required to complete tasks.

Moreover, the number of help needs impacts directly on the tool usability, so the researchers asked participants about their satisfaction.

Tool effectiveness means of the degree to which MVST is successful in decreasing effort time and producing the desired result of comprehensive. The effectiveness survey by participants shows 75% of participants Rate the MVST idea as 5, and 25% of them assess the tool as four value. This percentages is acceptance and indicates high satisfaction of MVST

CONCLUSION

This paper addressed the problem of software maintenance that considered as one of the most difficulties facing the maintainers, such as the lake of documentation and programmers changing. Researchers found software visualization the best way to speed up the maintenance process in an efficient way.

The main contribution in this thesis is to visualize unstructured Object-Oriented code in both static and dynamic views of Unified modeling language (UML), to make the program comprehensive more efficiently. This contribution creates a better chance for the maintainer of the understanding program quickly.

Another contribution is to produce new rules of extracting unstructured code, based on program behavior not only codes syntax "keywords". This contribution makes more probability of discovering relationships missed.

The last contribution is to facilitate perfective maintenance of existing static diagrams to reflect directly to the code.

REFERENCES

[1] . Goyal, G., & Patel, S. (2012). Importance of inheritance and interface in OOP paradigm measure through coupling metrics. Int. J. Appl. Inf, 4, 14-20.

[2] . Merino, L., Ghafari, M., Anslow, C., & Nierstrasz, O. (2018). A systematic literature review of software visualization evaluation. Journal of Systems and Software, 144, 165-180.

[3] . Raibulet, C., Fontana, F. A., & Zanoni, M. (2017). Model-driven reverse engineering approaches A systematic literature review. IEEE Access, 5, 14516-14542.

[4] . Briand, L., Labiche, Y., & Leduc, J. (2004). Towards the reverse engineering of UML sequence diagrams for distributed, multithreaded Java software. Carleton University, TR SCE-04-04, 1-85.

[5] . Martinez, L., Pereira, C., & Favre, L. (2014). Recovering sequence diagrams from object-oriented code: An ADM approach. Paper presented at the 2014 9th International Conference on Evaluation of Novel Approaches to Software Engineering (ENASE).

[6] . Améndola, F., & Favre, L. M. (2013). Science & engineering software migration: Moving from desktop to mobile applications. Paper presented at the V International Conference on Computational Methods for Coupled Problems in Science and Engineering (España, 17 al 19 de junio de 2013).

[7] . Bruneliere, H., Cabot, J., Dupé, G., & Madiot, F. (2014). Modisco: A model driven reverse engineering framework. Information and Software Technology, 56(8), 1012-1032.

[8] . Garzón, M. A., Lethbridge, T. C., Aljamaan, H., & Badreddin, O. (2014). Reverse engineering of object-oriented code into umple using an incremental and rule-based approach. Paper presented at the Proceedings of 24th Annual International Conference on Computer Science and Software Engineering.

[9] . Fawareh, H. J. (2016). Reverse Program Analyzed with UML Starting from Object Oriented Relationships. International Journal of Computer Science and Information Security, 14(3), 40.

[10] . Yang, J., Lee, Y., Gandhi, D., & Valli, S. G. (2017). Synchronized UML diagrams for object-oriented program comprehension. Paper presented at the 2017 12th International Conference on Computer Science and Education (ICCSE).

[11] . www.Structured-parsing.wikidot.com, last visit 20th of July 2019

[12] . www.Javacc.org, last visit 20th of July 2019

[13] . www.PlantUML.com, last visit 20th of July 2019

[14] . Emmerik & Waddington, "Using a decompiler for real-world source recovery", 11th Working Conference on Reverse Engineering, Delft, Netherlands, Netherlands 2004.

[15] . Bézivin, J. (2001). "From object composition to model transformation with the MDA". TOOLS USA 2001: Software Technologies for the Age of the Internet, 39th International Conference & Exhibition, Santa Barbara, CA, USA, July 29 - August 3, 2001.